Docket No: URI.5474

# UNITED STATES PATENT APPLICATION

*of*

**AUGUSTUS K. UHT**

**DAVID MORANO**

***and***

**DAVID KAELI**

***for***

# AUTOMATIC AND TRANSPARENT HARDWARE CONVERSION OF TRADITIONAL CONTROL FLOW PREDICATES

# AUTOMATIC AND TRANSPARENT HARDWARE CONVERSION OF TRADITIONAL CONTROL FLOW PREDICATES

## CROSS REFERENCE TO RELATED APPLICATIONS

This application claims priority from the provisional application designated serial number 60/198,300, filed April 19, 2000 and entitled "Automatic and Transparent Hardware Conversion of Traditional Control Flow to Predicates". This application is hereby incorporated by reference.

## TECHNICAL FIELD

The invention relates to the field of computing devices, and in particular to a computing device that includes automatic and transparent hardware conversion of traditional control flow predicates.

## BACKGROUND OF THE INVENTION

Computer programs typically use traditional control flow constructs to determine when and if instructions in the program are executed. Such constructs include *"if—then—else"* statements and various looping statements such as: *"while (condition is true){ ... }"*, *"for(i initialized to 1; while i<10; increment i every loop iteration){ ... }"* and *"do i=1 to 10 ... enddo"*. The majority of such control statements are realized with machine-level instructions called branches, and most of these are *conditional* branches.

Branches are used as follows. Most computers employ a model of computation using a pointer to the code of the program it is executing. The pointer is provided by a program counter (PC) that contains the address of the machine instruction the computer is currently executing. Every time an instruction is executed, the default action is to increment the program counter to point to the next instruction to be executed. Most useful programs employ branches to conditionally modify the contents of the program counter to point to other places in a program, not just the next instruction. Therefore, a conditional branch has the semantics: *if (condition is true) then load the program counter with a (specified) value.*

A well-known alternative to conditional branches is the use of predicates. A predicate is typically a one-bit variable having the values true or false; it is usually set by a comparison instruction. In this model every instruction has a predicate as an additional input. The semantics is that the instruction is only effectively executed (i.e., its output state changed) if the predicate is true. An example of equivalent classic control flow and modern predication is as follows.

| Classic code: | Predicated code: |
|---|---|
| 1. if (a==b) { | 1. Prod = (a==b); //Prod set to true if a equals b. |
| 2. z=x+y; | 2. IF (Pred) THEN z=x+y; //Operations performed only |
| 3. w=a+b; } | 3. IF (Pred) THEN w=a+b; // if Pred true. |
| 4. // later instructions: // all dependent on 1. | 4. // later instructions: NOT dependent on 1. |

In traditional computers, all instructions following a branch are dependent on the branch and must wait for the branch to execute before executing themselves. This has been demonstrated to be a significant barrier in realizing much parallelism within a program, thus keeping performance gains low.

However, with predication, only the instructions having the equivalent predicate as an input are dependent on the branch-remnant (the comparison operation). In the example and, in general, this means the instructions after the predicated instructions are now independent of the branch-remnant and may be executed in parallel with instructions before the branch-remnant, improving performance.

Current approaches to using predication use visible and explicit predicates. The predicates are controlled by the computer user and they use storage explicitly present in the computer's instruction set architecture (similar to regular data registers or main memory). They are explicit since there is at least a single 1-bit predicate hardware register associated with each instruction. The most extreme example of this is the IA-64 (Intel Architecture-64 bits) architecture. The first realization of this architecture is the Itanium (formerly Merced) processor, due to be on the market in the year 2000. Itanium has 64 visible-explicit predicate registers. See for example the document by the Intel Corporation, entitled "*IA-64 Application Developer's Architecture Guide*". Santa Clara, California: Intel Corporation, May 1999. Order Number: 24188-001, via www.intel.com. The predicates cannot be effectively used when the processor executes traditional IA-32 (x86) machine code. Therefore, billions of dollars of existing software cannot take advantage of Itanium without modification. Other types of microprocessors have similar constraints to x86 processors. That is, predicates are not currently in their instruction set, so they cannot take advantage of predication techniques.

It is possible to predicate just a subset of the instructions of a processor, but then the benefits of predication are much less. Full predication is preferred.

In prior work we devised a method for realizing an equivalent to full predication called minimal control dependencies (MCD). See for example, the papers by: (i) A. K. Uht, "Hardware Extraction of Low-Level Concurrency from Sequential Instruction Streams", PhD thesis, Carnegie-Mellon University, December 1985, available from University Microfilms International, Ann Arbor, Michigan, U.S.A; (ii) A. K. Uht, "An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code," in *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, University of Hawaii, in cooperation with the ACM and the IEEE Computer Society, January 1986; and (iii) A. K. Uht, "A Theory of Reduced and Minimal Procedural Dependencies," *IEEE Transactions on Computers*, vol. 40, pp. 681-692, June 1991. Each of these papers is incorporated herein by reference. MCD produced substantial performance gains, especially when coupled with another performance-enhancing technique of ours called disjoint eager execution, disclosed in the paper by A. K. Uht and V. Sindagi, entitled "Disjoint Eager Execution: An Optimal Form of Speculative Execution," in *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28),* pp. 313-325, IEEE and ACM, November/December 1995. This paper is also incorporated herein by reference. MCD can be considered to have *hidden* and *implicit* predicates, in that the predicates are not visible to the user, nor are they explicitly present in the processor. However, MCD has disadvantages when compared to predication such as a high hardware cost (e.g., more logic gates and storage) with relatively complex hardware. In particular, *j-by-j* diagonal bit matrices are required, where *j* is the number of instructions in the instruction window (those instructions currently under consideration for execution by the processor). In a high-ILP machine, $j$ might be 256 or more, leading to a cumbersome 32,000 or more bit diagonal matrix. Further, all of the bits need to be accessed and operated on at the same time, leading to a very complex and potentially slow hardware layout. Lastly, setting the contents of the matrix when instructions are loaded into the processor is also costly and potentially slow.

Therefore, there is a need for an automatic and transparent hardware conversion of traditional control flow predicates.

## SUMMARY OF THE INVENTION

Briefly, according to an aspect of the present invention, a computing device that provides hardware conversion of flow control predicates associated with program instructions executable within said computing device, detects the beginning and the end of a branch domain of said program instructions, and realizes the beginning and the end of said branch domain at execution time, for selectively enabling and disabling instructions within said branch domain.

These and other objects, features and advantages of the present invention will become apparent in light of the following detailed description of preferred embodiments thereof, as illustrated in the accompanying drawings.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a pictorial illustration of various branch arrangements;

FIG. 2 is a block diagram illustration of predicate-assignment hardware;

FIG. 3 illustrates a hidden-explicit predication example for disjoint branches;

FIG. 4 illustrates a hidden-explicit predication example for nested branches;

FIG. 5 illustrates a hidden-explicit predication example for overlapped branches; and

FIG. 6 illustrates a hidden-explicit predication example for mixed branches.

## DETAILED DESCRIPTION OF THE INVENTION

Hidden-explicit predicates are realized by the invention; the predicates are not visible to the user and thus may be implemented in any processor architecture, and the predicates occupy explicit hardware register bits in the processor, reducing cost and complexity. There are two parts to the invention: the predicate-assignment part, taking place when instructions are loaded into the processor, and the predicate-use part, taking place at instruction execution time.

Nomenclature: A branch's domain includes the instructions occurring between the branch and the branch's target. Thus, the branch controls the execution of the instructions within its domain. If the branch's condition evaluates true the branch is taken, and the instructions in its domain are not executed. If the branch's condition evaluates false, the branch is not taken and the instructions in the branch's domain are allowed to execute.

Multiple branch domains can be arranged in a number of different ways, each of which are combinations of the three basic arrangements: disjoint, nested and overlapped, as shown in FIG. 1. For full predication all possible combinations of these arrangements must be handled correctly. The invention does this.

Key Ideas: The predicate-assignment hardware detects the beginnings and ends of branch domains. The predicate-use hardware employs this information to realize the beginnings and ends of domains at execution time, performing the appropriate enabling and disabling of instructions in domains.

In general, as each new domain is encountered during code execution, a new condition is placed on the execution of the code within the new domain. If the branch condition of the domain's branch is $bc_i$, and the predicate of the code before the branch is $p_r$ then the effective predicate $p_e$ of the new code in the new branch's domain is computed as:

$$p_e = \overline{bc_i} \cdot p_r$$

When a domain is exited (upon reaching the corresponding branch's target instruction) the effect of the corresponding branch must be nullified, in other words $bc_i$ should have no effect on the execution of the following code. This is achieved by effectively OR-ing the opposite value of the branch condition with the current predicate; in other words, the following is effectively computed for the code after the branch domain:

$$p_{e2} = p_e + (bc_i \cdot p_r) = (\overline{bc_i} \cdot p_r) + (bc_i \cdot p_r) = p_r$$

This logic is realized by the combined operation of the predicate-assignment and predicate-use hardware.

## Predicate-Assignment Hardware and Operation

The predicate-assignment hardware assigns predicate and canceling predicate addresses to instructions as they are loaded into the processor's load buffer and before the buffer contents are sent to the instruction window for execution. The assignment is performed by detecting domain entries (branches) and exits (targets). The basic hardware structure is a branch tracking stack or buffer as shown in FIG. 2. FIG. 2 is a block diagram illustration of predicate-assignment hardware, that includes a stack that is associatively addressed by the current value of the *ilptr* (instruction lead pointer). The predicate address of the branch corresponding to a target address match with the *ilptr* is output from the hardware and used to augment the state of the instruction being loaded. The $p_r$ register holds the address of the current region's predicate. $p_r$ may point to the predicate from either a branch or a branch target.

In the context of the present invention, the term "stack" is used in its generic sense; it is contemplated that any kind of temporary storage may be used.

Each entry (row) in the stack corresponds to one branch. Typically, but not necessarily, a branch is on the stack only while the instruction load pointer *ilptr* value is within the branch's domain. The following fields compose each entry:

1. address of predicate corresponding to the branch $p_b$;

2. address of canceling predicate corresponding to the branch $cp_b$; in practice this may be derived from the branch's predicate address, so no explicit entry would be needed for canceling predicate addresses;

3. target address of the branch $ta_b$; and

4. valid bit flag $v_b$; true while the target of the corresponding branch has not yet been reached; the stack entry may be reclaimed and reused when the valid bit is false.

A branch is placed on the stack when it is encountered by the *ilptr* and is removed when its target is reached. In the case of overlapped branches, the target for a branch may be reached before a prior branch's target has been reached. In this case the overlapped branch has its valid bit flag $v$ bit cleared, and is removed from the stack when convenient.

The comparators look for a match between the instruction load pointer *ilptr* and the target addresses. If there is a match, it indicates that the instruction just loaded is the target of the matching branch (multiple matches will be considered later). The current canceling predicate address $cp_T$ is set equal to the canceling predicate address of the matching branch. The current canceling predicate address $cp_T$ is entered into the canceling predicate address field of the instruction being loaded.

Out-of-Bounds Branches: Branches with targets inside the window have been considered. It is also possible that a branch in the window may jump to a point not yet encountered by the predicate-assignment hardware. Therefore, the hardware illustrated in FIG. 2 is augmented with additional circuitry to handle these out-of-bounds branches. The new circuitry includes primarily another set of comparators for performing associative lookups on field "p".

The technique is as follows. A candidate branch for execution supplies its predicate address to the tracking buffer circuitry. The address is used as a key to perform a lookup on the "p" field. If a branch's domain is wholly contained in the window, then the branch will not have a valid entry in the buffer. Therefore, if the candidate branch does obtain a valid match, it is an out-of-bounds branch. The branch's target address is then read from the corresponding TA tracking buffer entry. The latter reduces storage costs, as target addresses need not be stored in the window, and also simplifies operation because target addresses do not need to be read from the window.

**Predicate-Use Hardware and Operation**

The Predicate-Use (PredU) hardware augments the state and operations of instructions held in the processor's instruction window. None of the Predicate-Use hardware is visible to the user (i.e., it does not appear in the processor's instruction set architecture) and thus may be applied to any type of processor.

The overall effect of the Predicate Use hardware is to chain predicate sources and sinks so as to both enforce the functionality of the system and to keep the hardware cost low. The alternative to chaining the predicates is to have many predicate inputs for each instruction, which would be costly in terms of additional instruction state and therefore also more complex in operation.

The Predicate Use hardware and operations differ depending on whether the instruction is a branch or an assignment statement. Both cases are now considered.

Branch PredU Hardware and Operations: The output predicates are evaluated or re-evaluated whenever the input predicate or branch condition becomes available or changes value, resp.

Input: $p_r$ - predicate of region, same as input predicate $p_{in}$.

Outputs:

branch predicate:

$$p_{out} = \overline{bc} \cdot p_r$$

branch canceling predicate:

$$cp_{out} = bc \cdot p_r$$

$bc$ is the Branch Condition of this branch, and has the values true (1) and false (0). It is set as the result of some comparison test operation such as: $A < B$. The comparison may be performed either as part of the branch's execution or as part of a prior instruction, depending on the processor architecture.

Execution Enabling Predicate: The branch executes whenever its inputs are available or change value. Therefore, all branches in the instruction window may execute in parallel and out-of-order.

Assignment Statement PredU Hardware and Operation: Assignment statements also have predicate inputs and outputs. These are used both for predicate-chaining and predicate-canceling. Recall that predicate-canceling occurs when a branch domain is exited.

Inputs:

$p_r$ - predicate of region; same as input predicate $p_{in}$ ; and

$cp_T$ - canceling predicate of targeting branch, if any; same as $cp_{in}$ .

Output:

$$p_{out} = p_r + cp_T = p_{in} + cp_{in}$$

$p_{out}$ is computed independently of the rest of the assignment statement's execution and computations.

Execution or Assignment Enabling Predicate: $p_I$ - same as output:

$$p_I = p_{in} + cp_{in}$$

The assignment instruction may modify its traditional sinks when $p_I$ is true. Such sinks are the results of the regular operations of the assignment statement, e.g., if the instruction is: $A = B + C$ then $A$ is a traditional sink and is written if the instruction's predicate evaluates true.

Case: Multiply-Targeted Instructions

There is a not-so-special case that can often arise in code and that we have not yet addressed. This is the case when an instruction is the target of more than one branch. In this scenario the hardware as described so far will not work, as it is only suitable for an instruction being the target of no more than one branch.

There are two solutions that can be employed to handle the multiple-target case. The first is to provide multiple canceling predicate fields for each instruction. This will cost more, but may be suitable for a small number of canceling predicates. However, we must handle the case when an instruction is the target of many branches (this is possible in many machines, although perhaps not likely).

A second solution is to insert a dummy No-Op instruction into the window after the current instruction if the instruction runs out of canceling predicate fields. The No-Op's canceling predicates can then be used in addition to the original instruction's. Since any number of No-Ops can be inserted, any number of branches targeting the same instruction can be handled. Of course, a price is paid for a "wasted" slot in the instruction window for each No-Op instruction added.

A suitable number of canceling predicate fields for one instruction may be empirically determined. It is likely that both solutions will be used in a typical processor.

Case: Branch is a Target of Another Branch

It is also possible, if not likely, that code will contain a branch that is the target of another branch. This scenario is readily handled by employing all of the predicate and canceling predicate logic in the branch, such that it appears as BOTH a branch and an assignment statement. The canceling predicate output of such an instruction is the same as that of an un-targeted version of the branch. The predicate output combines the functions of the branch predicate and the assignment statement predicate, with the branch portion using the assignment portion as its region predicate input:

$$p_{out} = \overline{bc} \cdot (p_r + cp_T) = \overline{bc} \cdot (p_{in} + cp_{in})$$

This works because the assignment portion effectively (logically) takes place before the branch.

Examples

We now present four examples to illustrate the operation of the hidden-explicit predicate system. The examples cover the following cases:

1. two disjoint branches, FIG. 3 (also covers the cases of straight-line code and a single branch);

2. two nested branches, FIG. 4.

3. two overlapped branches, FIG. 5.

4. three branches with a combination of nesting and overlapping, FIG. 6.

All of the examples have the same format. In the code column: "I" instructions are assignment statements, and "B" instructions are branches. The branch domains are shown with arrowed lines. Each example can be followed by first going down the *predicate-assignment* table entries, in order, as the instructions would be loaded. Using the tracking stack, this results in the predicate addresses shown in the $p_{in}$ and $cp_{in}$ columns being generated and entered into the corresponding instruction's fields in the instruction window.

Next, the predicate-use table entries may be examined to see how the predicates are evaluated at run-time, how their values are chained and how branch domains are effectively exited. For an example of the latter, refer to FIG. 3 and look at the $p_I$ entry for the assignment instruction at address 400. Although it has predicate inputs, their values cancel each other out, $p_I$ is effectively "1" and thus the instruction is always enabled for execution, as far as branches are concerned. This is correct, since it is outside the domains of all of the branches in the code example.

Although the present invention has been shown and described with respect to several preferred embodiments thereof, various changes, omissions and additions to the form and detail thereof, may be made therein, without departing from the spirit and scope of the invention.

What is claimed is: